A Framework for Migration from Legacy Software to Grid Services

Bartosz Baliś[1,2], Marian Bubak[1,2], Michał Węgiel[1]

[1] Institute of Computer Science, AGH, al. Mickiewicza 30, 30-059 Kraków, Poland
[2] Academic Computer Centre – CYFRONET, Nawojki 11, 30-950 Kraków, Poland
{balis,bubak}@uci.agh.edu.pl, mwegiel@student.uci.agh.edu.pl

Abstract. Porting legacy software to grid services platform is of paramount importance both from scientific and commercial point of view. Nonetheless presently no comprehensive design patterns facilitating this process are available. Existing approaches are limited to web services model and thus fail to meet requirements imposed by Open Grid Services Infrastructure. This paper presents a framework devised specifically for adaptation of legacy libraries and applications to grid services environment. We focus on designing a versatile solution and place particular emphasis on compliance with fundamental grid requirements. The proposed architecture is evaluated against such aspects like security, scalability, flexibility and efficiency. In addition, we provide characterization of a prototype implementation of our solution and indicate directions of its future development.

Keywords: legacy software, grid services, OGSI

1 Introduction

The grid environment can be treated as a federation of heterogeneous, distributed and dynamic resources which are seamlessly and transparently integrated across diverse virtual organizations and collaborate in order to deliver the desired quality of service [1]. Entities constituting the grid are not subject to centralized control and are coordinated by means of standard, open, general-purpose protocols and interfaces. The fundamental paradigms which lay the foundations of grid computing are **resource virtualization** and **service orientation**.

The semantics of grid services and their interaction model are defined by Open Grid Services Infrastructure [5] and Architecture [6], respectively. OGSI extends the functionality of web services by imposing additional requirements concerning such aspects as naming, security and lifetime management. Generally speaking, grid service can be thought of as a special case of web service which provides a set of well-defined interfaces and follows specific conventions.

OGSI augments web services in three main areas. Firstly, there is a potentially dynamic and transient nature of grid services (instances can be created and destroyed on demand). Each non-permanent service is assigned the finite lifetime. When it expires the service is automatically removed via the soft-state

destruction mechanism. Secondly, grid services can comprise not only methods but also attributes (in a sense this resembles the paradigm of object oriented approach). And finally, there is a notion of secure (involving authentication and authorization) and reliable method invocation.

Obviously, grid computing needs to satisfy numerous requirements characteristic of any large-scale, distributed technology. Specifically, issues of performance, security and scalability deserve a detailed insight. Apart from these, we should consider flexibility and manageability as well as the ease of administration and system maintenance.

One of the key challenges facing the process of migration to grid services technology is the necessity of retaining interoperability with legacy applications. This is due to the fact that significant corporate investments in the existing high quality and validated software have to be preserved during transition to the new platform. From economic perspective it is essential to ensure that legacy applications can be incorporated into the grid infrastructure without involving much development effort. A cost-effective migration strategy is a prerequisite for the anticipated wide adoption of grid services technology.

The purpose of this paper is to outline the concept of a framework designed to facilitate the process of adaptation of legacy software to grid services environment. In our discussion we aim at devising a universal and comprehensive solution, applicable to a broad range of diverse real life situations.

2 State of the art

The related research is present both in scientific and commercial settings. In [2] a proposal of a semiautomatic technique for conversion of legacy C interfaces to their Java equivalents is presented. Two auxiliary tools: JACAW and MEDLI are introduced which allow for code wrapping and data mapping, respectively. Although separate, they are intended to work in collaboration. The former exploits Java Native Interface in order to bridge between C and Java code. It provides facilities which enable to generate Java interfaces from the given C header files. MEDLI allows for conversion of the obtained Java code to components that can be then deployed in the Triana-based grid environment. Triana is a workflow composer which can be used to build grid services from pluggable components.

This approach is connected with two major limitations. Most importantly, it is restricted to configurations in which legacy applications are located on the same machine as the service container. It stems from the fact that the communication is realized by means of JNI. In a vast array of situations this inflexibility can prove unacceptable. Secondly, it is constrained to Java language as far as hosting environment is concerned. Along with the emergence of different, non-Java OGSI implementations this shortcoming may become problematic.

In [3] a conceptual architecture for adaptation of legacy applications to web services technology is presented. Three components constitute the essence of the proposed solution: web service container, web service adapters and backend legacy servers. Each web service is associated with a certain adapter which is

responsible for connecting to the appropriate backend server(s) on behalf of the client. The role of adapters is to hide the complexity of calling backend functions which typically involves the communication through proprietary protocols.

Since this approach is primarily focused on web services (as opposed to grid services) it fails to meet a significant number of requirements such as already mentioned issues of security and lifetime management. The most important disadvantage is its inherent insecurity. Each backend server demands an open port on which it can listen to the clients' requests. In complex installations this may create serious security vulnerabilities. Another drawback connected with the above architecture is its inflexibility. Service adapters have to be configured statically with regard to the locations of the corresponding backend servers so that the communication can be established. In consequence the infrastructure lacks such features like dynamic reconfiguration in case of component failure or the automatic process migration between computing nodes.

3 Proposed architecture

The general approach that we employ while porting a certain legacy application to grid services technology is based on the separation of its interface and implementation. Legacy interface is first extracted, then refined and finally exposed in the form of a grid service. Legacy implementation is encapsulated into a software wrapper which constitutes the necessary adaptation layer. The resulting components can be deployed on disparate hosts.

3.1 Overall system structure

In the proposed architecture we can distinguish three major components: **backend host**, **hosting environment** and **service client** (as depicted in Fig. 1). They are potentially located on different machines and communicate by means of SOAP messages. The only way in which information can be exchanged between the components is remote method invocation performed on grid services residing in the hosting environment. No other communication protocol is ever used.

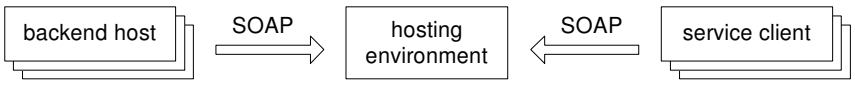

Fig. 1. Architecture

Hosting environment is basically the container for grid services which provides the needed middleware facilities. Backend host represents the machine on which legacy software is installed. In order to enhance performance and reliability we may decide to deploy several instances of the same legacy application on

different computing nodes. Thus we can end up having multiple backend hosts associated with a single service.

Since client-side interaction is performed in a standard fashion, we will not elaborate it in more detail. Instead, let us now concentrate on the cooperation between backend hosts and the container. As we stated earlier, in the context of a particular service there can be multiple backend hosts, possibly in diverse geographic locations. Such redundancy improves scalability and fault-tolerance which both are essential for delivery of the required quality of service. We will assume that on each backend host there is a copy of appropriate legacy application installed. The question we need to address is how to organize the communication within the resulting configuration.

One of the most straightforward solutions is to make backend hosts act as servers to which the container forwards user requests using some legacy protocol. However, this approach is connected with a number of disadvantages. First and foremost, it is insecure due to the open ports introduced on backend hosts. Such vulnerability can prove unacceptable in the grid environment. Another drawback of this model is that the identity of backend hosts cannot be established and in effect there is no guarantee that we cooperate with the trusted machines. Again, it is a disqualifying feature in case of security-sensitive applications. Furthermore, in this approach it is necessary either to implement our own security mechanisms or to rely on their legacy equivalents. In both cases there is no compliance with standard grid authentication and authorization procedures. Yet, issues concerning security are not the only shortcomings of the solution being analyzed. Another one is its inherent inflexibility. There is the need of manual configuration on the container-side with respect to backend hosts to which it can connect. Moreover, there is no efficient way of determining which servers are available at the specified point in time, especially when the configuration embraces a large number of backend hosts in highly dynamic environment. This may lead to the significant degradation of performance since the container will spend plenty of time investigating to which server it can delegate the particular client request. Taking all these facts into consideration it is clear that we have to adopt different approach.

According to our solution, backend hosts do not play the role of servers any more. On the contrary, they resemble service clients. In addition to this, they are expected to register their capabilities of participation in the processing of client requests. They volunteer to accept tasks only when they possess sufficient resources to complete them within reasonable amount of time. We will demonstrate that this approach is free of all the above mentioned limitations. A comprehensive characterization of its advantages will be presented later on. At the moment we will only summarize its most important features.

First of all, it is secure. There are no open ports introduced on backend hosts. Legacy applications are no longer required to be accessible from the outside. It is also possible to obtain the identity of backend hosts because as clients they employ grid service method invocation mechanism which conveys caller's cre-

dentials. Furthermore, the configuration is flexible and dynamic. The container automatically keeps track of the pool of backend hosts.

3.2 Backend host

Let us now turn our attention to the concepts lying behind the abstraction of backend host. This component plays a crucial role in the whole infrastructure since it is responsible for actual request processing. As we discussed earlier, it constitutes an environment in which legacy software is being executed.

As illustrated in Fig. 2, each backend host comprises two flavors of processes: **master** and **slave** ones. They together form the adapter layer which encapsulates the legacy interface. The advantage of using processes instead of threads is greatly reduced development effort since concurrency problems are no longer of concern.

Master process serves two fundamental purposes. It is responsible for host registration and creation of slave processes as and when required. There is only one master process per host. Slave processes are in charge of direct cooperation with legacy system. It is worth mentioning that this collaboration can have various forms and range from library function calls to communication over prioprietary protocols. Slave processes perform two primary tasks: method call translation and data structure mapping. Each such process corresponds to a single client.

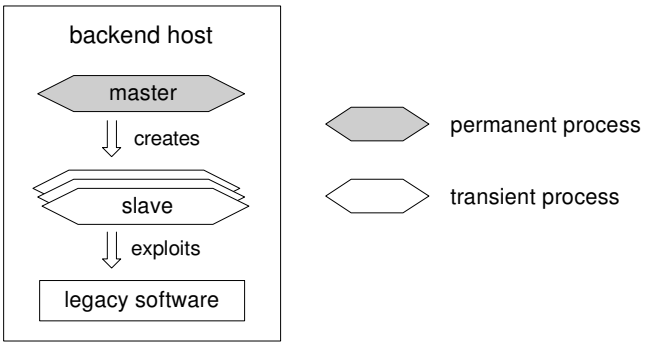

Fig. 2. Backend host

One thing certainly worth emphasizing by this occasion is that master process is a permanent entity whereas each slave process is transient and has limited lifetime. As clients come and go the number of concurrently running slave processes changes respectively.

Both types of processes need to communicate with hosting environment in order to fulfill their function. From this perspective they can be perceived as service clients because they utilize SOAP messages for conversation. However

they cooperate with dedicated grid services which are inaccessible for ordinary clients.

3.3 Hosting environment

Let us now take a closer look at a collection of grid services which up to this point were referred to as the hosting environment component. They jointly serve one principal purpose which is shielding the clients from the interaction with backend hosts.

The central concept is presented in Fig. 3. For each legacy system that is exposed in the form of a grid service we deploy three permanent services: **registry**, **factory** and **proxy factory**. Depending on the number of clients exploiting our system we may have varying number of transient services which are **instance** and **proxy instance**. This stems from the fact that each client is expected to create a separate service instance for its exclusive usage.

Another classification of the considered services can be based on their accessibility as seen from client's point of view. We differentiate between externally visible and hidden services. The former are exemplified be factory and instance. All other services are designated for internal purposes and are not available to clients.

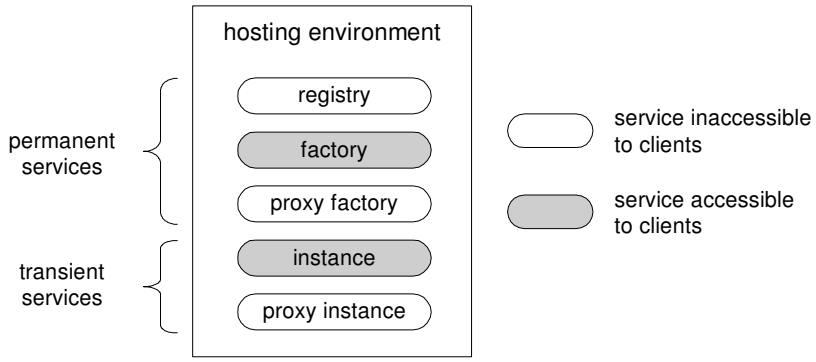

Fig. 3. Hosting environment

To get a better sense how the whole architecture works, we will now provide a rationale for the consecutive services explaining their rudimentary roles.

Registry is a service recording backend hosts which offered their participation in request processing. It is devised as a notification source. Backend hosts are supposed to express their consent to accept tasks by subscription to notifications generated by registry. This in turn is obliged to assign each pending client to the chosen backend host by sending the appropriate notification.

The remaining services can be grouped into two pairs, each consisting of factory and the corresponding instance. We discriminate two kinds of factories:

ordinary and proxy ones. The former are meant for clients in order to enable them for dynamic creation of instances. The latter are used to construct auxiliary service instances which are responsible for mediation between backend hosts and actual clients. The reason for this separation is cosmetic: we are trying to avoid mixing internal and external interfaces. However this is not the only justification for this decision. Besides this, some technical problems connected with soft-state destruction arise when both pairs are merged.

4 Operational scenarios

Having examined the overall system architecture, we yet need to concentrate on typical scenarios that take place within the confines of standard client-service interaction. More detailed analysis of system behavior in various circumstances will give us more thorough understanding of individual components as well as the infrastructure as a whole.

4.1 Initialization

Let us now follow the course of action that takes place as a part of a start-up procedure. From the client's perspective, initialization involves only one activity: calling factory to create a new service instance. On internal system side, however, there are a number of steps to be performed. They are schematically depicted in Fig. 4. The essential prerequisite for this scenario is an operative environment with permanent services already set up. It is also assumed that at least one master process is registered (i.e. subscribed to notifications) in the context of the service being considered.

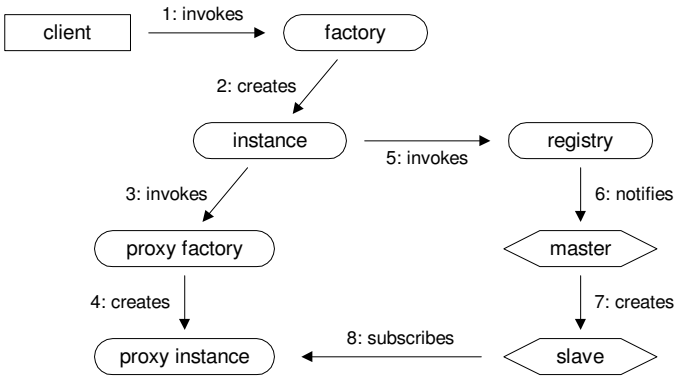

Fig. 4. Initialization

The procedure of initialization is triggered by the client via the request of service instance creation. The whole scenario is in principle based on callback

methods connected with lifetime management. They are invoked by the container in order to signal such events like instance construction or destruction. Thanks to this mechanism, it is possible to execute custom actions upon the creation of service instance. This is exactly what happens in our scheme, beginning with the third step. In the first place, proxy factory is contacted to create the corresponding proxy instance. Then, registry service is invoked. As a result of this call a certain master process is selected and notified about the awaiting client. Following this, a slave process is spawned. It automatically subscribes itself to the proxy instance in order to receive notifications about method calls carried out by the client.

At this point, there are two issues which deserve further elaboration. The first one is how to choose the backend host from the pool of registered machines. In fact we can employ various algorithms to accomplish this task. The most straightforward approach is first-in first- out queue. However it seems that it would be preferable to exploit additional information gathered during host registration - for example computational capabilities of individual machines. Secondly, we need to guarantee that the newly created slave process will subscribe to notifications produced by the appropriate proxy instance. In other words, we have to supply it with the address of the created proxy. This information has to be sent as a part of a notification to the selected master process.

From theoretical point of view, the most elegant solution would be to use grid job submission mechanism. We would no longer need host registration procedure and master processes. Instead, we would basically submit slave processes to the grid. Nonetheless, in practice, this approach brings many limitations. First of all, in order to submit the job we have to specify the location (or endpoint) of the appropriate grid service. In current Globus Toolkit implementation it implicitly implies the machine on which the task will be actually executed. Furthermore, on that particular host there must be the container installed (in order to host job submission service). It is obvious that we do not want to deploy the hosting environment on each machine that happens to be backend host from time to time.

These two deficiencies have a profound impact on system scalability and flexibility. In cases when we have a huge volume of backend hosts in highly dynamic environment the solution based on job submission can fail to meet our requirements. We might experience an intolerable overhead when searching for the most appropriate backend host for a specific task. At a particular point in time only a fraction of backend machines may be able to process requests. Thanks to registration we have continually the most up to date picture of the system. There is no need of polling in order to determine the available resources.

4.2 Method invocation

In this section we will be exploring the details of the most critical part of the architecture. Effective scenario of method invocation is crucial to the system performance since it is the most frequently occurring event.

After successful initialization we should expect the configuration to look like as illustrated in Fig. 5. The three transient entities shown: instance, proxy instance and slave process can be thought of as an exclusive property of the corresponding client. Slave process is subscribed to the notifications generated by proxy instance.

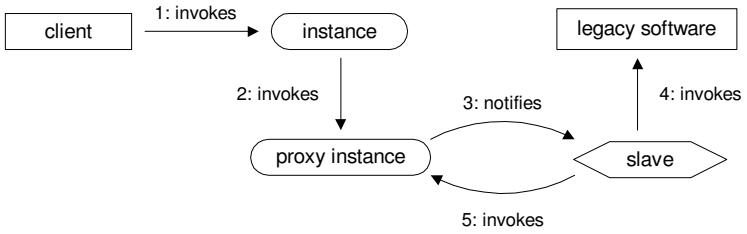

Fig. 5. Method invocation

Let us follow the sequence of steps which constitute the whole scenario. Whenever the client invokes a method, its complete description, together with arguments that were passed on, is forwarded to the proxy instance. The request is then further propagated to the slave process by means of notification delivery. On receipt of such notification the slave process translates the method invocation to the respective function call and transfers the control to legacy application. When processing is done the obtained results are supplied to the proxy instance via separate asynchronous method invocation. This in turn causes the blocked method to return to the service instance and after that to the client from which it originated.

One remaining aspect we need to address in our discussion is the analysis of performance overhead introduced by additional layers of indirection. The communication between instance and proxy instance takes place within the borders of a single container so it should not be a bottleneck. Local calls can be highly optimized as compared to remote ones. The latter occur as a result of cooperation between proxy instances and slave processes. There are exactly two remote SOAP invocations per each client call: first for notification delivery and second for supplying the results returned by legacy application.

4.3 Destruction

Grid services support two distinct modes of instance deletion called explicit and soft- state destruction, respectively. The former is performed on client's demand. The latter resembles garbage collection in that it is executed when the instance lifetime has expired or inactivity time limit has been exceeded. In case of both modes we are supposed to reclaim the occupied resources when the container invokes the appropriate callback method.

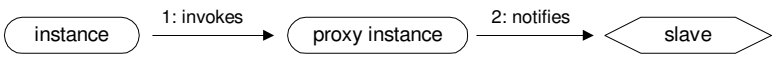

Fig. 6. Destruction

Destruction scenario, presented in Fig. 6, is relatively simple. The only task we have to carry out is to forward the received removal request to the proxy instance. Once it is delivered, the slave process is notified that it should terminate its execution. Proxy instance will be removed by the container shortly afterwards by means of soft-state destruction.

5 System characteristics

After thorough inspection of individual components and scenarios comprising the proposed architecture it is time to examine its higher-level features. We will characterize the presented solution from different perspectives and confront it with the requirements that need to be satisfied in the grid environment.

5.1 Security

We have already mentioned two aspects concerning the security of our infrastructure. Firstly, there are no permanently open incoming ports introduced on backend hosts. Secondly, we are able to check the identity of the registered machines. We owe both these advantages to the fact that backend hosts act as clients rather than servers. Now we will provide more comprehensive discussion and cover the details of authentication and authorization procedures.

In principle, we rely on Grid Security Infrastructure and employ X.509 certificates. One important feature of these is that they allow us to assign identity both to users and hosts. This distinction will be important during authorization. Communication integrity and privacy are guaranteed by mechanisms of message signing and encryption, respectively.

Access to all services is restricted to subjects holding adequate identities. We have separate authorization procedures for external (visible to clients) and internal (hidden) services. The former is based on user certificates. Access is granted to clients eligible to use the particular service. The latter uses host certificates. In this case we accept only those machines which are permitted to register in the context of a specific service.

Thus, the complete security configuration can be thought of as two lists of identities (or distinguished names). First one for clients which can use our service and second one for hosts which can register at this service. In effect, maintenance of the security policy should not involve much administrative effort.

One more point deserving further remarks is subscription mechanism. According to the specification, notification sink is required to expose a network accessible endpoint. In the context of our proposal this means that backend hosts should allow incoming connections in their ephemeral port range. In case

of security sensitive applications this may prove unwelcome as we may want to close all ports except from outgoing ones. Under such circumstances we can resign from notifications altogether and simulate their functionality with blocking method invocations. Although this solution may involve greater development effort, it is as good as our original approach in terms of performance and even better as regards security. Taking these facts into consideration we will examine the whole concept more closely. General idea is pretty simple. We replace the subscription mechanism by repetitive blocking method calls which return only when subsequent notifications should be delivered. Let us illustrate it with an example. Suppose that we have already passed through the initialization procedure and a certain slave process is associated with the appropriate proxy instance. Evidently, this process is in charge of two primary tasks: intercepting client calls and supplying the corresponding results. Both of them can be realized without notifications. Whenever slave process is ready to process next client request it basically calls a special method. This invocation blocks until the client performs some action. Having detected that client call was made, the full description thereof is returned to slave process by previously blocked call. As soon as processing is finished the results are delivered by means of another method call performed by slave process. And then the cycle is repeated. It is clear that we still have only two SOAP messages exchanged per single client call. Thus, this approach does not incur any additional communication overhead and at the same time considerably improves security.

5.2 Scalability

We can enumerate several factors that jointly enable to achieve high scalability of our architecture. First of all, actual processing is always delegated to backend hosts. This means that computations can be highly distributed. Moreover, service instances residing in the container never consume excessive resources. Their activity is limited to request and response forwarding. Furthermore, backend hosts volunteer to accept tasks only when their load is reasonable. This supports automatic load balancing as well as resource reservation which are essential for the assurance of the specific level of quality of service. Thanks to delegated control we earn high responsiveness to temporal peaks in resource utilization. This in turn is crucial for on-demand computing.

5.3 Flexibility

The proposed solution offers a considerable degree of flexibility. There are a few reasons for this. Most notably, processes are not bound to backend hosts. They can change their locations even during execution (job migration) as long as security constraints are not violated. In addition, due to the fact that we employ registration model, the architecture is immune to unexpected changes of configuration, like component or network failures. There is no need of polling in order to determine operative nodes.

5.4 Other properties

Undoubtedly, an important advantage of our architecture is that legacy software can remain unchanged in place where it was initially installed. It is not necessary to move programs between hosts or to perform any additional endeavors.

Another thing worth emphasizing is the independence of any particular implementation of hosting environment. In general we make no assumptions about the language on which the container platform is based.

The framework was designed to accommodate a wide variety of legacy applications regardless whether they work as servers or are only system libraries inaccessible from other hosts. In this respect our model is fairly universal. It is noteworthy that the whole system can be viewed as a multi-tier architecture. Business logic is delegated to backend hosts, while the container constitutes only the service presentation layer, making them available for clients.

6 Project status

In this section we will briefly summarize a pilot implementation of the described approach. The main concept presented in this paper was partially implemented in the project aiming at moving grid application monitoring system, OGM-G [4], to grid services platform. We used Globus Toolkit 3.0 [7] together with gSOAP package [8] and GSI plug-in [9] for this purpose. The main architecture was alike the one described here. It was recognized that the method we employed was quite universal and that it can be generalized and used in other similar ventures.

Our current work is primarily focused on performance measurement. We are also developing additional test cases in order to observe how this infrastructure behaves in various circumstances. The direction in which the project is expected to evolve is the development of tools facilitating the use of the proposed framework. These include automatic generation of interfaces and stub routines. In further phases of the project we plan to design and implement several such utilities.

References

1. Foster, I., Kesselman, C., Nick, J., Tuecke, S.: The Physiology of the Grid
2. Huang, Y., Taylor, I., Walker, D., Davies, R.: Wrapping Legacy Codes for Grid-Based Applications
3. Kuebler, D., Eibach, W.: Adapting Legacy Applications as Web Services (http://www-106.ibm.com/developerworks/webservices/library/ws-legacy)
4. Baliś, B., Bubak, M., Funika, W., Szczepieniec, T., Wismüller, R., Radecki, M.: Monitoring Grid Applications with Grid-enabled OMIS Monitor
5. Open Grid Services Infrastructure: http://www.gridforum.org/ogsi-wg/
6. Open Grid Services Architecture: http://www.globus.org/ogsa/
7. The Globus Project: http://www.globus.org
8. The gSOAP Project: http://www.cs.fsu.edu/~engelen/soap.html
9. The GSI Plugin Project: http://sara.unile.it/~cafaro/gsi-plugin.html